Apache Performance Tuning
Part Two: Scaling Out

Sander Temme
sander@temme.net

June 29, 2006

Abstract

As your web site grows in popularity, you will get to the point when one server doesn't cut it anymore. You need to add more boxes, and this session discusses several approaches to scaling out. We will cover webserver load balancing, SSL offload and separating application tiers. We will also discuss configuring the Apache HTTP Server to front Apache Tomcat servers and how to load balance between Tomcat servers. Finally, we will cover Java VM and database tuning.

1 Introduction

Building out a web server infrastructure is a large and multifaceted challenge. The server infrastructure for any large web site is necessarily customized for the needs and requirements of that site, thus it is very difficult to make valid general statements about scaling technologies. This paper and its accompanying ApacheCon presentation give a general overview of the field, touching upon approaches and technologies rather than discussing them in depth.

1.1 Why Would You Scale Out?

Scaling Out is a business decision. You may scale out because you cannot meet your performance goals with a single web server. Alternatively, you may scale out to meet reliability and uptime goals. There are many approaches to scaling out, with varying price tags. So whatever your motivation, to scale out your web infrastructure you will have to justify added expenses for server hardware, network equipment, possibly software licenses and maintenance contracts, and most certainly system administration time and resources.

2 Building Out: Load Balancing

Scaling Out means adding more servers. The primary issue that arises when servers are added is how to direct client transactions to multiple hosts. The user

does not know and does not need to know that multiple servers are in use. They just want to point their browser to `www.example.com` and spend a lot of money on your products or services. In this section we will review several techniques to distribute client transactions across your hosts.

2.1 Load Balancing with DNS

You have a great deal of control over where your users direct their transactions by using the Domain Name Service (DNS) for your site. This seems obvious, but it is critical to scaling. When your users connect to `www.example.com`, they don't care to which IP address this resolves. If you can manipulate this resolution, you can send the user to whichever physical server you prefer, which can be of great benefit to the infrastructure

2.1.1 Distinct Servers for Distinct Services

One way to distribute transaction load across multiple physical servers is to give each server a separate task. For your `www.example.com` site, use an `images.-example.com` server to serve static image content, a `secure.example.com` server to handle SSL transactions, etc. This approach allows you to tune each server for its specialized task. The downside is that this approach does not scale by itself: once, for instance, your secure server runs out of processing headroom, you will have to add more machines using one of the techniques described below.

2.1.2 DNS Round-Robin

If you operate multiple servers that perform identical functions, you can distribute client transactions among them using Domain Name Server Round-Robin. The principle behind this technique is that a single server hostname resolves to a different IP address from your server pool for each DNS resolution request. For instance, if you have three web servers with the IP addresses

```
10.11.0.113
10.11.0.114
10.11.0.115
```

and you have your name server return each of those addresses in turn for queries to your web server name (`www.scalingout.org`), roughly one third of all clients will connect to each of your web servers. Since popular name server implementations like *bind* implement this technique by default, it is very simple to implement without any resource requirements besides control over your DNS zone.

How "roughly" this works depends on many factors, over few of which you have any control. Client-side resolvers cache query reponses, as do intermediate nameservers at ISPs and corporations. Large ISPs and corporations represent many potential users, all of whom would be directed to the same web server for as long as their nameserver caches the original lookup. However, across your

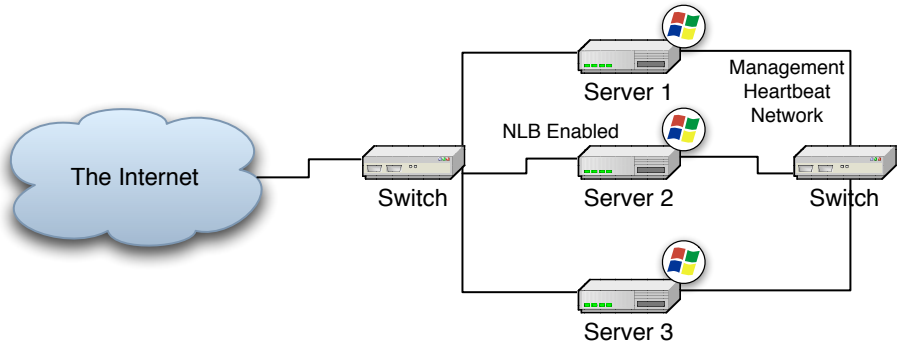

Figure 1: The *Network Load Balancing* feature is included in Windows Server System. It clusters servers as peers, using a separate network connection for load balancing decision and heartbeat traffic.

entire user population these discrepancies may even out. You can help this process by reducing the cache timeout for query results in your zone file. An example zone file that uses DNS Round-Robin is shown in Appendix A.

DNS Round-Robin as a load balancing approach is often disparaged because of its simplicity: it does not take into account the load on the servers, and can not compensate for server outage. If a server goes down for any reason, one third of all clients will still be directed to the nonfunctional server. If these considerations are important to you, consider one of the more sophisticated load balancing approaches described below. However, do not dismiss DNS Round-Robin out of hand. Depending on your requirements, it may be all you need.

2.2 Peer-based Load Balancing

You can turn a collection of individual servers into a cluster by using load balancing techniques. In this section we will discuss Microsoft's approach.

2.2.1 Windows Network Load Balancing

Windows Load Balancing Service (WLBS) technology has been available since Windows NT Server 4.0, Enterprise Edition and is now included in Windows Server 2003 under the name Network Load Balancing (NLB). Using Network Load Balancing, you can turn up to 32 servers into a cluster. The service work by having every machine assume the same IP address, and the same MAC address, on the clustered interface(s). Incoming connections arrive at all members of the cluster simultaneously from the network switch. The NLB software communicates between cluster members over a unicast or multicast backchannel and is responsible for the load balancing decisions. It sits between the network

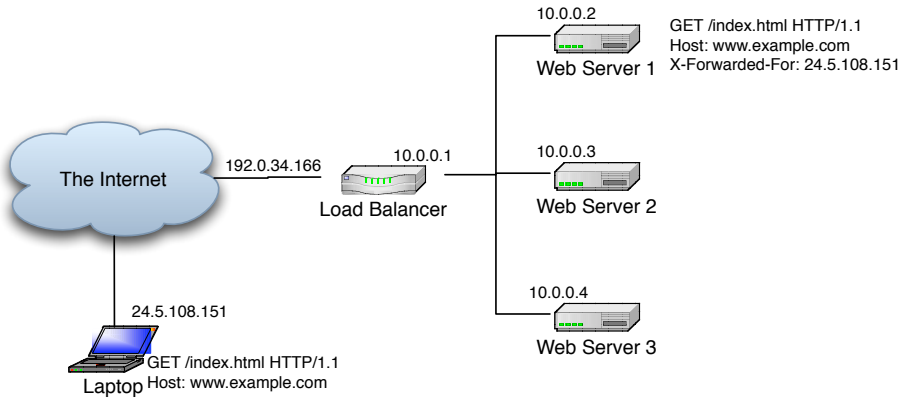

Figure 2: A Load Balancer Appliance forwards incoming requests to one of the web servers

card driver and the TCP/IP stack, and regulates which cluster member gets to answer each incoming request. Cluster members whose NLB module doesn't communicate with the other members get removed from the pool. This allows NLB to provide High Availability as well as load balancing functionality.

Because it operates below the TCP/IP layer, Network Load Balancing should be compatible with any service that runs on the server machines. Each cluster member has to be configured exactly the same. Please see your Windows Server 2003 documentation for details.

2.3 Load Balancing Appliance

The market for dedicated load balancing appliances is now quite crowded, with offerings from vendors like Cisco, F5, Juniper, Foundry and many others vying for your attention. These products can be pricy, but are powerful solutions for load balancing your server farm.

2.3.1 How a Load Balancer Works

Load balancing appliances or application switches sit between the web servers and the outbound network connection and intelligently distribute traffic across multiple web servers. They typically keep track of the load and availability of the servers, and adjust their load balancing decisions accordingly. Many of these products can operate on several layers of the network stack and can inspect incoming requests to make load balancing decisions based on source address, requested URI, cookies submitted etc.

2.3.2 Linux Virtual Server

The Linux Virtual Server project is an open source load balancing and high availability implementation. Its core module, IP Virtual Server, is included in the kernel as of version 2.6.10. Auxiliary software like `ipvsadm` is only an install away. If you are considering rolling your own load balancing solution, consider Linux Virtual Server.

The primary disadvantage of Linux Virtual Server is that it does not come as a nice, shiny plug-and-play box with a support contract. Instead, it looks more like an Erector Set[1] of bits and pieces that you get to integrate yourself. However, this disadvantage can also be a strength: it allows you to build a solution that best fits your needs. However, the absence of a 24x7 support plan may upset your decision makers. You can find an example configuration for Linux Virtual Server in Appendix B.

2.4 Load Balancing with Tomcat

The Tomcat application server and its mod_jk Apache HTTP Server connector provide load balancing functionality that allows running a number of Tomcat instances behind a single web server.

This approach to Scaling Out is important for both performance and reliability reasons. On the performance side, the web server can distribute resource intensive application traffic among multiple application servers. From a reliability point of view, it becomes possible to selectively take down Tomcat instances for maintenance, without affecting overall availability of your application. Or, you can periodically stop-start Tomcat instances to prevent the build up of issues like memory leaks. Additionally, in the event a Tomcat instance crashes, the load balancing mod_jk worker will automatically take it out of the pool until it becomes available again.

The load balancing functionality in mod_jk uses a round-robin algorithm to assign requests to its workers. It maintains *sticky sessions* based on the Tomcat session cookies, so requests that belong to the same application session will always be routed to the same Tomcat instance.

A complete example of a Tomcat/mod_jk load balancing configuration is available in Appendix C. You can also load balance a number of Tomcat servers using the mod_proxy_balancer module that comes with Apache 2.2. This approach is discussed in Section 2.5.

2.4.1 Session Replication in a Tomcat Cluster

Sticky sessions allow for session persistence across a load balancer by having it always route traffic that carries a specific session cookie to the same back-end Tomcat server. However, when one of the backj-end servers crashes or is taken out of service for maintenance, the sessions it is serving at that moment are lost. This is highy undesirable if these sessions represent customers' shopping carts

[1] Perhaps better known in Europe as *Meccano*.

or banking transactions. Tomcat clustering and session replication prevents session loss by copying session information from any cluster member to all the other members as soon as a session is created.

Clustering works over IP multicast. There are specific requirements for the environment and session objects, which must be serializable. These requirements are discussed in the Tomcat documentation[2]. An example configuration is discussed in Appendix C.

2.5 Load Balancing with Apache 2.2

Version 2.2 of the Apache HTTP Server includes a load balancing proxy module, `mod_proxy_balancer`. This add-on to `mod_proxy` can balance incoming requests between multiple back-end servers. It can persist connections to a particular back-end based on a configurable Cookie key like `JSESSIONID` or `PHPSESSIONID`. The configuration looks as follows:

```
Listen 80
LogLevel debug
TransferLog logs/access_log

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so

ProxyPass / balancer://mycluster/
ProxyPassReverse / http://localhost:16180
ProxyPassReverse / http://localhost:16280

<Proxy balancer://mycluster>
  BalancerMember http://1.2.3.4:80
  BalancerMember http://1.2.3.5:80
</Proxy>
```

The configuration above will equally distribute requests between two back-end web servers. If your back-end servers are running Tomcat, you can enable sticky sessions as follows:

```
...
<Proxy balancer://tccluster>
    BalancerMember http://localhost:15180 route=tc1
    BalancerMember http://localhost:15280 route=tc2
    BalancerMember http://localhost:15380 route=tc3
</Proxy>

<Location />
    ProxyPass balancer://tccluster/ stickysession=JSESSIONID
    Require all granted
</Location>
```

The server.xml file of each Tomcat server has to be edited to include a `jvmRoute` attribute in the `Engine` element. For instance, in the first Tomcat, you have:

[2] /tomcat-docs/cluster-howto.html on a default Tomcat installation

```
<!-- ... -->
<Server port="15105" shutdown="SHUTDOWN">
<!-- ... -->
  <Service name="Catalina">
    <Connector port="15180" maxHttpHeaderSize="8192"
               maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
               enableLookups="false" redirectPort="8443" acceptCount="100"
               connectionTimeout="20000" disableUploadTimeout="true" />
<!-- ... -->
    <Engine name="Catalina" defaultHost="localhost" jvmRoute="tc1">
<!-- ... -->
    </Engine>
  </Service>
</Server>
```

The `mod_proxy_balancer` module also contains a small management application, which you can enable using the following configuration snippet:

```
<Location /balancer-manager>
  SetHandler balancer-manager
  # Your access control directives here
  Order Allow,deny
  Allow from all
  # ...
</Location>
```

The management feature requires the presence of `mod_status` in addition to `mod_proxy_balancer`. As you can see, the load balancing reverse proxy in Apache HTTP Server 2.2 is quite easy to set up. For more information about the configuration options, please see the mod_proxy documentation[3]. More information on the algorithms for the load balancing decision can be found in the `mod_proxy_balancer` documentation[4].

3 Building Out: Separate Tiers

Most web applications can be separated into multiple distinct tiers:

1. Web server tier(Apache, IIS, Sun ONE)
2. Application server tier (Tomcat, PHP, WebLogic, etc.)
3. Database server tier (MySQL, Oracle, Postgres, etc.)

Every tier has distinct and particular performance requirements. Moving each tier to their own hardware allows you to tune and scale them individually. The fact that all of the individual applications communicate with each other over TCP/IP already makes this move even easier.

[3] http://httpd.apache.org/docs/2.2/mod/mod_proxy.html
[4] http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

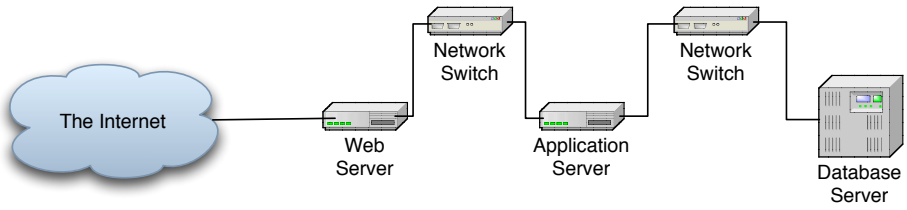

Figure 3: When you divide your application functionality into multiple tiers, you can scale out by adjusting each tier for maximum performance, independent of the other tiers.

3.1 The Web Server Tier

The **Web Server** tier communicates directly with the users. It is responsible for maintaining connection with a wide variety of client browsers across potentially slow and far-flung connections. This causes a markedly different load on the operating system TCP stack than the long-lived, local, high speed connections between web and application server, and between application server and the database. The web tier can also be configured to serve the application's static content: HTML pages, images, JavaScript, etc. It passes only the requests for dynamically generated content (PHP scripts, JavaServer Pages, RSS feeds) on to the application tier. The type of server used for this tier typically has one or two CPUs and enough memory to fit the requisite number of httpd processes. Storage is not a concern.

3.2 The Application Server Tier

The **Application Server** tier generates all dynamic content. It receives requests from the web tier and maintains connections to the database tier. The operating system can be tuned specifically to run an application server platform such as a Java virtual machine. The type of server used for this tier may have multiple CPUs as required to run application threads. These servers have more memory than the web servers as required by the application platform, but storage is not important on this tier.

3.3 The Database Server Tier

The **Database Server** tier stores all application data. The application server tier connects to the database tier using the JDBC protocol or native database libraries. Database access can be a considerable bottleneck for application performance, so performance is an important consideration for this tier. The type of server used for this tier should have sufficient CPU power and RAM to run the database application, and come with scalable, redundant storage like a RAID-5 array.

4 Designing Your Site for Scaling Out

4.1 Designing for a Load Balancer

A Load Balancer introduces an additional moving part to your web server infrastructure. While most load balancer solutions do their best to appear transparent to the application, you may find some issues that you can solve by properly designing your application.

The main issue arises with session persistence. The HTTP protocol is inherently stateless, which is great for a load balancer: it can consider each incoming request for itself and make a completely independent load balancing decision based on its criteria. Session persistence potentially complicates this issue, especially if a user's session exists only on the server that initially created it. If a subsequent request from that user is directed to a different back-end server, the session is lost. Most load balancing solutions solve this problem by consistently directing requests from a particular IP address to the same back-end server. Some can inspect incoming HTTP requests and make load balancing decisions based on session cookies.

These load balancer based fixes should be enough under most circumstances, but your requirements may be more stringent: what if the user reconnects after his lunch break and the load balancer has timed out because of inactivity from the user's IP address? Or the user reconnects from a different IP address (let's say she leaves one Starbucks and reconnects from the one across the street)? Or the server that holds the user's session goes offline because of a crash or maintenance? If it is important to you to maintain user sessions under circumstances like these, you should build session persistence into your application. Users sessions are likely to cause more reads than writes. You could write session information to your backend database, or use a special, fast database with a write-through cache just for session maintenance.

5 Conclusion

Scaling out your web site is a mixed blessing. While you get to serve more transactions and, presumably, do more business, the additional hardware, software and network segments will also give you more intricacies to oversee. You get to manage, maintain en secure a farm of servers instead of just one. The configuration of your servers, and application software and content design will be highly influenced by the infrastructure design decisions you make, and they will be heavily intertwined. However, with judicious planning, scaling out can be an efficient and effective solution to increased site demands.

A DNS Round-Robin Zone File

The following is a very basic DNS Zone file that uses Round-Robin DNS to balance three web servers.

```
scalingout.org. 86400 IN SOA ns.scalingout.org. sctemme.scalingout.org. (
           2006051401 ; Serial
           86400      ; refresh (1 day)
           7200       ; retry  (2 hours)
           8640000    ; expire (10 days)
           86400 )    ; minimum (1 day)

scalingout.org.     IN      NS      bagheera.scalingout.org.

gw              IN      A       10.11.0.1
bagheera        IN      A       10.11.0.2

; ...

mail            IN      CNAME   bagheera
ns              IN      CNAME   bagheera

www             IN      A       10.11.0.113
                IN      A       10.11.0.114
                IN      A       10.11.0.115
```

B Linux Virtual Server Configuration

This example uses a Linux Virtual Server director running Ubuntu 5.10 (The Breezy Badger). The outside interface of the Director has IP address 10.0.0.1, its inside interface is on 192.168.1.1. Two back-end web servers are connected to an internal interface of the Director. Their Ethernet interfaces are configured for 192.168.1.2 and 192.168.1.3 respectively, and both have 192.168.1.1 for default gateway. On the Director machine, the file `/etc/ipvsadm.rules` has the following information:

```
# ipvsadm.rules
-A -t 10.0.0.1:80 -s rr
-a -t 10.0.0.1:80 -r 192.168.1.2:8080 -m -w 1
-a -t 10.0.0.1:80 -r 192.168.1.3:8080 -m -w 1
```

and the file /etc/defaults/ipvsadm looks as follows:

```
# Do not edit! Use 'dpkg-reconfigure ipvsadm'.
AUTO="true"
DAEMON="none"
```

The tool mentioned in the comment has interactive menus for the two variables. This is all the configuration necessary to run Linux Virtual Server in NAT mode: a reboot or the command `/etc/init.d/ipvsadm start` issued as root starts the load balancer.

C Example Tomcat 5.5 Load Balancing Configuration

This example uses an Apache HTTP server with the mod_jk module, and three Tomcat instances load balanced with sticky sessions. Note that the example does not cover any access control or authorization. It is recommended that you restrict access to the mod_jk status Worker.

The following is an httpd.conf snippet that sets up mod_jk:

```
# Load the mod_jk module. You would obviously use the
# path to your own httpd installation.
LoadModule jk_module /Volumes/Files/asf/httpd-r415210w/modules/mod_jk.so

# Mount your web applications. We are using the
# examples that come with Tomcat
JKMount /servlets-examples/* loadbalancer
JKMount /*.jsp loadbalancer

# The mod_jk Manager application. Please implement appropriate
# access control on production systems.
JkMount /jkmanager/* jkstatus

# Log mod_jk activity. You probably want a less verbose log level
JKLogFile logs/jk_log
JKLogLevel debug

# The JKWorkerProperty directive is new as of mod_jk 1.2.7. It allows
# you to specify mod_jk worker configuration directives in httpd.conf
# instead of a separate workers.properties file
JKWorkerProperty worker.list=loadbalancer,jkstatus

# Define three Tomcat instance workers
JKWorkerProperty worker.tc1.port=15109
JKWorkerProperty worker.tc1.host=localhost
JKWorkerProperty worker.tc1.type=ajp13
JKWorkerProperty worker.tc1.lbfactor=1
JKWorkerProperty worker.tc2.port=15209
JKWorkerProperty worker.tc2.host=localhost
JKWorkerProperty worker.tc2.type=ajp13
JKWorkerProperty worker.tc2.lbfactor=1
```

```
JKWorkerProperty worker.tc3.port=15309
JKWorkerProperty worker.tc3.host=localhost
JKWorkerProperty worker.tc3.type=ajp13
JKWorkerProperty worker.tc3.lbfactor=1

# Define a load balancer worker that uses the three
# Tomcat instances
JKWorkerProperty worker.loadbalancer.type=lb
JKWorkerProperty worker.loadbalancer.balance_workers=tc1, tc2, tc3

# Define the mod_jk status worker
JKWorkerProperty worker.jkstatus.type=status
```

In the server.xml configuration file of each Tomcat, you need to set the jvmRoute attribute of the Engine element to match the corresponding worker name. For instance for the first Tomcat instance:

```
...
    <Engine name="Catalina" defaultHost="localhost" jvmRoute="tc1">
...
```

If you don't set this attribute, sticky sessions will not work!

C.1 Session Replication in a Tomcat Cluster

The default `server.xml` configuration file that comes with Tomcat contains an example clustering configuration. This example is commented out. To enable clustering, find the `<Cluster> ... </Cluster>` element in the configuration file and uncomment the entire element. If your test Tomcats are running on the same host, change the *tcpListenPort* attribute of the *Receiver* element to a unique value for every instance. For instance:

```
        <Cluster className="org.apache.catalina.cluster.tcp.SimpleTcpCluster"
                 managerClassName="org.apache.catalina.cluster.session.DeltaManager"
                 expireSessionsOnShutdown="false"
                 useDirtyFlag="true"
                 notifyListenersOnReplication="true">

<!-- ... -->
            <Receiver
                className="org.apache.catalina.cluster.tcp.ReplicationListener"
                tcpListenAddress="auto"
                tcpListenPort="15101"
<!-- ... -->
```

The second Tomcat instance gets `tcpListenPort="15201"`, the third 15301 etc. Every web application that uses session replication has to have a *distributable* element in its web.xml file. For instance, to make the Tomcat Servlet examples distributable, edit *webapps/servlets-examples/WEB-INF/web.xml* for every cluster member and add `<distributable />` to the `<web-app>` container.

C.2 Testing your Deployment

To test the deployment as described above, place an *index.html* file in the *webapps/ROOT* directory of every Tomcat instance with some distinctive text like "This is Tomcat 1" (and 2, and 3 respectively). Subsequent requests from a client to your load balancing Apache should show you the index pages from all your back-end serves in succession.

Testing sessions, sticky sessions and replication is not as simple. Browsers very efficiently manage session cookies in a fashion completely transparent to the user, so it is not easy to observe the exact behaviour between client and server. For this reason, it is often better to use a command-line client like curl(1). For instance, you can test session behaviour using the following command[5]:

```
$ curl -i -b /tmp/cookiejar -c /tmp/cookiejar \
> http://localhost:15080/servlets-examples/servlet/SessionExample
```

The first time you invoke this command, you will see a `Set-Cookie:` header appear in the response, and the generated HTML page will display the session ID. The last element of the session ID should be the *jvmRoute* value of the Tomcat instance that received the request. Subsequent requests will show the same session ID in the page, since requests will be routed to the same instance using the sticky session. Store some data in the session by invoking:

```
$ curl -i -b /tmp/cookiejar -c /tmp/cookiejar \
> http://localhost:15080/servlets-examples/servlet/SessionExample\
> ?dataname=foo\&datavalue=bar
```

The value *foo = bar* shoud now appear under *The following data is in your session* in the generated HTML page. Shut the receiving Tomcat down by calling its *bin/shutdown.sh* script, and try the request again. If you didn't configure clustering, you will receive a response from a different Tomcat instance with a new session ID. If you did configure clustering, however, the new Tomcat will have replicated the session data from the first one and you should see the same session ID, and the same key/value pair stored in the session.

[5] This example requires that you have the Tomcat Servlet examples installed.